# Коректност на итеративни алгоритми

Тодор Дуков

## Какво имаме предвид под коректност?

За целите на този курс един алгоритъм ще наричаме **коректен**, ако:

1. завършва при всеки вход
2. връща правилен изход при всеки вход

*Забележка.* Въпреки че ние ще имаме това разбиране в курса, в реалния свят нещата не винаги се случват така:

- разглеждат се алгоритми, които могат и да не завършват за някои входове – от теоритична гледна точка са интересни за хората, които се занимават с теория на изчислимостта
- разглеждат се алгоритми, които много често (но не винаги) връщат правилния вход – обикновено това се прави с цел бързодействие

## Едно *“ново”* понятие

Специално за итеративните алгоритми се въвежда ново понятие - **инварианта**. Човек може да си мисли за инвариантата като на друго име на индукцията:

- тук отново си имаме база
- индукционното предположение и индукционната стъпка се обединяват в *“нова”* фаза, наречена **поддръжка**
- довършителните ще наричаме **терминация**

**Внимание.** Това, за което се използват инвариантите, е да се докаже коректността на ЕДИН итеративен цикъл, не на цял алгоритъм. Когато в алгоритъма ни има няколко цикъла, на всеки от тях трябва да съответства по една инварианта.

## Един простичък пример

Нека разгледаме следния алгоритъм за степенуване на 2:

```
1  int pow2(int n) // тук n ще бъде положително
2  {
3      int result = 1;
4
5      for (int i = 0; i < n; ++i)
6      {
7          result *= 2;
8      }
9
10     return result;
11 }
```

**Инварианта.** *При всяко достигане на ред 5 имаме, че* $\texttt{result} = 2^{\texttt{i}}$.

**База.** Наистина при първото достигане на ред 5 имаме, че $\texttt{i} = 0$ и от там $\texttt{result} = 1 = 2^{\texttt{i}}$.
**Поддръжка.** Нека при някое непоследно достигане твърдението е изпълнено. Тогава във следващото достигане на цикъла на $\texttt{i}$ присвояваме $\texttt{i} + 1$, и след това на $\texttt{result}$ присвояваме $\texttt{result} * 2$, като знаем, че преди $\texttt{result}$ е бил $2^{\texttt{i}_{\texttt{old}}}$. Така е ясно, че $\texttt{result}$ ще стане $2^{\texttt{i}_{\texttt{old}}+1} = 2^{\texttt{i}}$.
**Терминация.** Ако се намираме в последното достигане на ред 5, то тогава $\texttt{i} = \texttt{n}$, откъдето ще върнем $\texttt{result} = 2^{\texttt{n}}$.

# С инвариантите трябва да се внимава

Един от често срещаните капани, в които попадат хората, е да не си формулират инвариантата добре. Много е важно инварианта да дава достатъчна информация за това което наистина се случва в алгоритъма. За целта ще разгледаме един пример:

```
1  int selection_sort(int *arr, int n)
2  {
3      for (int i = 0; i < n - 1; ++i)
4      {
5          int min_index = i;
6
7          for (int j = i + 1; j < n; ++j)
8          {
9              if (arr[j] < arr[min_index])
10                 min_index = j;
11         }
12
13         int temp = arr[i];
14         arr[i] = arr[min_index];
15         arr[min_index] = temp;
16     }
17 }
```

На интуитивно ниво е ясно какво прави кода. Намира най-малкия елемент, и го слага на първо място. След това намира втория най-малък елемент, и го слага на второ място, и т.н.
Нещо, което някои биха се пробвали да направят за първия цикъл, е следното:

*При всяко достигане на ред 3 подмасивът* $\texttt{arr}[\texttt{0} \ldots \texttt{i} - 1]$ *е сортиран.*

Проблемът с това твърдение, е че може много лесно да се измисли алгоритъм, за който това твърдение е изпълнено, и изобщо не сортира елементите в масива:

```
1  int trust_me_it_sorts(int *arr, int n)
2  {
3      for (int i = 0; i < n; ++i)
4      {
5          arr[i] = i;
6      }
7  }
```

Очевидно този за този алгоритъм горната инварианта е изпълнена, но той е безсмислен. Получаваме сортиран масив, но за сметка на това губим цялата информация, която сме имали за него.
Нещо друго, което е важно да се направи, е първо да се формулира инварианта за вътрешния цикъл, и после за външния, като тънкият момент тук е, че ще ни трябват допускания за първата инварианта. Идеята е, че външния цикъл разчита на вътрешния да си свърши работата, и обратно вътрешния разчита (не винаги) на външния преди това да си е свършил работата.
Тук може да се направи следното (доказателството остава за упражнение на читателя):

**Инварианта** (вътрешен цикъл)**.** *При всяко достигане на ред* 7 *имаме, че* `min_index` *е индексът на най-малкия елемент в масива* $\texttt{arr}[\texttt{i} \ldots \texttt{j} - 1]$.

**Инварианта** (външен цикъл)**.** *При всяко достигане на ред* 3 *имаме, че масивът* $\texttt{arr}[\texttt{0} \ldots \texttt{i} - 1]$ *съдържа сортирани първите* `i` *по-големина елементи на* `arr`*, като останалите се намират в* $\texttt{arr}[\texttt{i} \ldots \texttt{n} - 1]$.

Обикновено в доказателството на коректност на алгоритми най-трудното е да се формулира инвариантата. Ако човек има добре формулирана инварианта, доказателството и на първо място възможно, а на второ – по-лесно.

# Задачи

*Задача* 1. Да се:

1. напише алгоритъм, който сумира числата в един масив
2. докаже неговата коректност
3. изследва сложността му по време и памет

*Задача* 2. Даден е следният алгоритъм:

```
bool alg(int *arr, int n)
{
    for (int i = 0; i < n - 1; ++i)
    {
        for (int j = i + 1; j < n; ++j)
        {
            if (arr[i] == arr[j])
                return true;
        }
    }

    return false;
}
```

1. Какво връща той? Отговорът да се обоснове.
2. Каква е неговата сложност по време и памет?

*Задача* 3. Даден е следният алгоритъм:

```
int fib(int n) // n ще бъде поне 0
{
    if (n < 2)
        return n;

    int a = 0, b = 1;

    for (int i = 1; i < n; ++i)
    {
        int temp = a;
        a = b;
        b = temp + b;
    }

    return b;
}
```

Да се докаже, че `fib(n)` връща `n`-тото число на Фибоначи.

*Задача* 4. Даден е следният: алгоритъм:

```
void mult(int **A, int **B, int **C, int n)
{
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            int cell_sum = 0;

            for (int k = 0; k < n; ++k)
            {
                cell_sum += A[i][k] * B[k][j];
            }

            C[i][j] = cell_sum;
        }
    }
}
```

Да се докаже че при вход $\mathtt{n} \times \mathtt{n}$ матрици `A`, `B` и `C`, функцията `mult(A, B, C, n)` записва в `C` произведението на `A` и `B`. Да се намери сложността му по-време и памет.